Neural Network Action Policy Verification via Predicate Abstraction

Marcel Vinzent, Jörg Hoffmann
Saarland University
Saarland Informatics Campus
Saarbrücken, Germany
{vinzent, hoffmann}@cs.uni-saarland.de

Abstract

Neural network (NN) action policies are an attractive option for real-time action decisions in dynamic environments, as a single call to the NN suffices to obtain the next action. Such an application obviously requires a high degree of trust in the NN. But how to gain such trust? The ultimate solution would be to *verify* the behavior induced by the policy. This is potentially very hard as it compounds the state space explosion with the difficulty of analyzing even single NN decision episodes. Here we make a first step towards this challenge. We show how to compute predicate abstractions of the policy state space subgraph induced by fixing an NN action policy. We consider safety i.e. the question whether the policy may end up in unsafe states when started from a given set of safe states. We implement the approach leveraging off-the-shelf tools for answering NN-satisfiability questions. We conduct a feasibility study in Racetrack. The results indicate that NN policy verification may be more feasible than it first appears.

Introduction

Neural networks (NN) are an increasingly important representation of action policies. In basic AI research, huge successes were achieved in game playing (Mnih et al. 2013; Silver et al. 2016, 2018), and an increasing body of work is dedicated to learning action policies in planning (Issakkimuthu, Fern, and Tadepalli 2018; Groshev et al. 2018; Garg, Bajpai, and Mausam 2019; Toyer et al. 2020). Within and beyond basic research, a growing trend is to rely on NN action policies for real-time decision taking in dynamic environments. The vision is elegant and simple: a single call to the NN policy suffices to obtain the next action.

Yet this vision comes with high requirements on trust in the neural network. How to gain such trust? There are various possible avenues, including for example any manner of explainable AI that may help to elucidate the NN's action decisions; or shielding, which augments the NN action policy with a safety guard (e.g. (Könighofer et al. 2017; Alshiekh et al. 2017; Fulton and Platzer 2018)). Here we address *verification* of the behavior induced by the policy. This is potentially very hard as it compounds the state space explosion with the difficulty of analyzing even single NN decision episodes. Here we make a first step towards this challenge.

Copyright © 2021, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

We consider *safety* in the following form: *Given an NN action policy π and a set of states S_0 known to be safe, is an unsafe state (that satisfies an unsafety condition) reachable from S_0 under π?* Apart from the NN itself, there are two sources of complexity in this question: (i) the number of start states $|S_0|$ when S_0 is specified in a compact representation, e.g., a formula over the state variables; and (ii) the number of states reachable from a single $s \in S_0$ when state transitions, or π itself, are non-deterministic. Recent work (Gros et al. 2020) addresses (ii) via statistical model checking, which is however unable to address (i) as start states must be explicitly enumerated. Here we instead concentrate fully on (i), simplifying matters by assuming deterministic transitions and policy choices.

We choose *predicate abstraction* as a well-established method for abstract verification (Graf and Saïdi 1997; Ball et al. 2001; Henzinger et al. 2004). Predicates here are linear constraints over the state variables (e.g. $x = 7$ or $x \leq y$), and abstract states are characterized by truth value assignments to a set of predicates $\mathcal{P}$ (grouping together all concrete states that result in the same assignment). Like in other abstraction methods common in planning (e.g. (Edelkamp 2001; Helmert et al. 2014; Seipp and Helmert 2018)), transitions are over-approximated to preserve all possible behaviors. Therefore, if π is safe starting from S_0 in the abstraction, then π is safe in the original (concrete) state space.

Observe that the latter statement does not actually necessitate to verify the full state space Θ, but rather only the *policy-restricted* state space Θ^π, i.e., the subgraph of the state space that results from fixing the action policy π. Hence we build predicate abstractions of Θ^π. We refer to these as *policy predicate abstractions* $\Theta^\pi|_\mathcal{P}$. We then check whether unsafe states are reachable from S_0 in $\Theta^\pi|_\mathcal{P}$. If this is not the case then π is proven to be safe.

The part of Θ^π reachable from S_0 will often be much smaller than Θ (provided S_0 itself is much smaller than Θ), especially under deterministic conditions. This gain is however counterbalanced by the need to analyze π, which arises when deciding whether there is a transition between two abstract states A and A' in $\Theta^\pi|_\mathcal{P}$. This is a *NN-satisfiability* problem: *Does there exist an input (a state) s to the NN π such that $s \in A$ and $\pi(s) = a$ and executing a in s results in a state $s' \in A'$?* Answering such questions over and over again can clearly become infeasible when the abstrac-

tion and/or the NN are large. On the positive side though, we can leverage a lot of progress on analyzing NN decision episodes here, from other research communities.

SMT solvers such as $Z3$ (de Moura and Bjorner 2008) can be used for exact NN-satisfiability tests. Much progress has already been made on devising solvers dedicated to NN analysis (Katz et al. 2017; Gehr et al. 2018). We design an algorithm computing $\Theta^\pi|_\mathcal{P}$, leveraging such solvers through a family of over-approximating easier satisfiability questions. In our implementation, we employ $Z3$ for exact tests as well as partial tests of various forms, and we employ *Marabou* (Katz et al. 2019) as an over-approximation that relaxes state variables to be continuous. We also implemented an iteration over *Marabou* calls forcing it to eventually identify an integer solution or prove that none exists.

We base our work on JANI, an automata-based language from model checking (Budde et al. 2017). This makes sense as the problem addressed is intrinsically linked to the formal methods community; also, automata-based languages are well suited to model exogenous agents (Hoffmann et al. 2020) and thus an important source of environment non-determinism (complexity source (ii) above, which we don't handle here yet but which clearly is important in the future).

We conduct a feasibility study in Racetrack (e.g. (Barto, Bradtke, and Singh 1995; Bonet and Geffner 2003)), evaluating scalability in the number of abstraction predicates and the number of start states $|S_0|$. The results give some indication that NN policy safety verification may be more feasible than it first appears, at least in the setting where (i) is the only source of complexity. *Marabou* picks up much of the size reduction of the policy-restricted state space Θ^π, at a feasible computational cost. That said, more experiments will be needed to assess the approach with confidence.

We note that our approach can ultimately serve not only for verification, but also for interactive visualization of NN action policy behavior, and thus a form of explainable AI. Policy predicate abstractions are in principle amenable to this purpose, with users seeing a small summary of policy behavior (namely $\Theta^\pi|_\mathcal{P}$) at any one time, and abstraction refinement methods providing the means to "zoom in". Exploring this possibility remains a topic for future work.

Networks of Automata

In this section, we outline the structure of the automata networks we consider. The detailed definition will be made available in an online technical report.

State Variables We consider automata networks, that are defined over a set of integer state variables $\mathcal{V}$. Each $v \in \mathcal{V}$ is associated with a *lower bound* $lo(v)$ and an *upper bound* $up(v)$ with $lo(v) < up(v)$, such that its domain D_v is the non-empty, bounded integer interval $\{lo(v), \dots, up(v)\}$.

A *(partial) variable assignment* $s_\mathcal{V}$ over a set of variables $\mathcal{V}$ assigns to each variable a value from its domain, i.e., a function with domain $dom(s_\mathcal{V}) \subseteq \mathcal{V}$ where $s_\mathcal{V}(v) \in D_v$ for all $v \in dom(s_\mathcal{V})$. If $dom(s_\mathcal{V}) = \mathcal{V}$, $s_\mathcal{V}$ is *complete*.

Given two variable assignments s_1, s_2, we denote by $s_1[s_2]$ the *function update* of s_1 by s_2, i.e., $dom(s_1[s_2]) = dom(s_1) \cup dom(s_2)$ with $s_1[s_2](v) = s_2(v)$ if $v \in dom(s_2)$ and otherwise $s_1[s_2](v) = s_1(v)$. Moreover, we say s_1 *agrees* with s_2 iff $s_1(v) = s_2(v)$ for all $v \in dom(s_1) \cap dom(s_2)$. If additionally $dom(s_2) \subseteq dom(s_1)$, we also write $s_2 \subseteq s_1$.

Expressions The automata networks involve linear integer expressions over $\mathcal{V}$. A **linear integer expression** can be written in the form $d_1 \cdot v_1 + \dots + d_r \cdot v_r + c$, with integer scalars $d_1, \dots, d_r, c \in \mathbb{Z}$ and integer variables $v_1, \dots, v_r \in \mathcal{V}$. We denote the set of linear integer expressions by Exp_{int}. Given $e_1, e_2 \in Exp_{int}$, $e_1 \bowtie e_2$ with $\bowtie \in \{\leq, =, \geq\}$ is a **linear integer constraint**. We denote the set of linear integer constraints and conjunctions thereof by Exp_{bool}. Finally, by $e(s_\mathcal{V})$ we denote the evaluation of $e \in Exp_{int} \cup Exp_{bool}$ over a variable assignment $s_\mathcal{V}$.

Automata Network Semantics Intuitively, in an automata network, a single automaton consists of a set of **locations** connected by **edges**. Each edge links a **source** location to a **destination** location. An edge can be taken, i.e., the automaton can transit from source to destination, only if its **guard**, a constraint from Exp_{bool}, evaluates to true over the current state variable assignment. While **silent** edges can be taken independently, **labeled** edges can only be taken as part of a **synchronization**. We consider synchronization based on **synchronization vectors**. A synchronization vector specifies for each automaton an edge label (or none if the automaton does not participate). Automata may synchronize taking labeled edges whose combination matches one of the synchronization vectors. If an edge is taken, the state variables are updated according to a set of **assignment updates** evaluated over the current state variable assignment. An assignment update is a tuple $(v, e_a) \in \mathcal{V} \times Exp_{int}$, where v is the *assigned variable* and e_a is an *update expression*.

Given the intuition above, we formalize the state space of an automata network as a labeled transition system (LTS) $\Theta = \langle \mathcal{S}, \mathcal{A}, \mathcal{T} \rangle$, where

- the **states** $\mathcal{S} = \mathcal{S}_{\mathcal{V}_{loc}} \times \mathcal{S}_\mathcal{V}$ are tuples of complete variable assignments over $\mathcal{V}_{loc}$ and $\mathcal{V}$, with location state variables $\mathcal{V}_{loc}$, i.e., for each automaton a variable assigned to the current location.
- there is an **action** $(g_{loc}, g, u_{loc}, u) \in \mathcal{A}$ for each silent edge and for each combination of automata-edge pairs that match a synchronization vector, where
  - the *source location constraint* g_{loc}, respectively the *location update* u_{loc}, is a partial variable assignment over $\mathcal{V}_{loc}$, assigning each participating automaton to the source location, respectively the destination location, of the edge it takes,
  - the guard $g \in Exp_{bool}$ is the conjunction of the guards of the taken edges,
  - *(variable) update* u is the union of assignment updates over all taken edges. We assume that u is well-defined, i.e., for each variable there is at most one assignment update, and denote the set of updated variables as $dom(u)$.
- for the set of **transition** $\mathcal{T} \subseteq \mathcal{S} \times \mathcal{A} \times \mathcal{S}$, it holds $((s_{\mathcal{V}_{loc}}, s_\mathcal{V}), (g_{loc}, g, u_{loc}, u), (s'_{\mathcal{V}_{loc}}, s'_\mathcal{V})) \in \mathcal{T}$ iff $g_{loc} \subseteq$

$s_{\mathcal{V}_{loc}}$, $g(s_{\mathcal{V}})$ evaluates to true, $s'_{\mathcal{V}_{loc}} = s_{\mathcal{V}_{loc}}[u_{loc}]$, and $s'_{\mathcal{V}} = s_{\mathcal{V}}[u(s_{\mathcal{V}})]$, where $u(s_{\mathcal{V}})$ denotes the partial variable assignment induced by u evaluated over $s_{\mathcal{V}}$. Here, we assume that $u(s_{\mathcal{V}})$ respects the variable domains.

NN Action Policies

Given the state space of an automata network, we are only interested in the behavior possible under a fixed action policy specifying which action is chosen in which state. More formally, given state space $\Theta = \langle \mathcal{S}, \mathcal{A}, \mathcal{T} \rangle$, an *action policy* π for Θ is a total function $\mathcal{S} \to \mathcal{A}$, and the policy-induced subgraph $\Theta^\pi = \langle \mathcal{S}, \mathcal{A}, \mathcal{T}^\pi \rangle$, where $\mathcal{T}^\pi = \{(s, a, s') \in \mathcal{T} \mid \pi(s) = a\}$, is the *policy restriction* of Θ under π.

Neural Networks We consider action policies implemented by *neural networks* (NN). Neural networks consist of *neurons*, computational units, that output the result of an *activation function* applied to a weighted linear combination of their inputs (Gros et al. 2020; Sarle 1994). How the weights and the biases are *learned* is beyond the scope of this paper, cf., e.g., *Q-learning* (Mnih et al. 2015).

The activation function is typically non-linear. In our current work, we restrict to *rectified linear units* (ReLU) (Nair and Hinton 2010), i.e., with activation function $max(0, x)$. ReLU is *piecewise-linear*, i.e., its domain can be divided into **finitely** many intervals on which it is linear ($x < 0$, $x \geq 0$). We also restrict to feed-forward networks. A feed-forward NN consist of an *input layer*, arbitrarily many *hidden layers*, and an *output layer*, each composed of a set of neurons. The inputs to each neuron are the outputs of all neurons of the previous layer and computational results are fed forward layer-wise from input to output. In our case, the input is a state $s \in \mathcal{S}$ and the output an action $a \in \mathcal{A}$. We say, the neural network implements an *NN action policy* $\pi \colon \mathcal{S} \to \mathcal{A}$.

The precise NN policy structure will be in the technical report. The restriction to feed-forward networks with piecewise-linear activation functions is important for the design of the NN-satisfiability queries required for the computation of the predicate abstraction of Θ^π (discussed later). Our framework is more generally applicable in principle however, and it remains a question for future work which more general classes of neural networks can be addressed.

Note that we do not assume that $\pi(s)$ is applicable in s, i.e., no $s' \in \mathcal{S}$ with $(s, \pi(s), s') \in \mathcal{T}$ may exist. This raises the issue of **deadlock verification** as known from concurrent systems (e.g. (Holzmann 2004)), which here takes the form of checking whether the policy may choose an inapplicable action on a reachable state and thus get stuck. Addressing this problem is an interesting topic for future work.

Policy Safety We now formalize our notion of policy safety, i.e., whether from a set of *start states* S_0 a state from a set of *unsafe states* S_U is reachable in the policy restricted state space Θ^π. In the context of automata networks, we compactly represent these sets by a *start condition* and an *unsafety condition* respectively, both composed of a location variable assignment and a logical constraint over $\mathcal{V}$.

Definition 1 (Policy Safety Property)**.** Let $\Theta = \langle \mathcal{S}, \mathcal{A}, \mathcal{T} \rangle$ be the state space of a network of automata and $\pi \colon \mathcal{S} \to \mathcal{A}$ an action policy for Θ. A *policy safety property* of Θ and π is a tuple $((s_{\mathcal{V}_{loc},0}, e_0), (s_{\mathcal{V}_{loc},U}, e_U))$ with location variable assignments $s_{\mathcal{V}_{loc},0}, s_{\mathcal{V}_{loc},U}$ over $\mathcal{V}_{loc}$ and constraints $e_0, e_U \in Exp_{bool}$. We call $(s_{\mathcal{V}_{loc},0}, e_0)$ the *start condition* and $(s_{\mathcal{V}_{loc},U}, e_U)$ the *unsafety condition*.

Θ^π is *unsafe* with respect to $((s_{\mathcal{V}_{loc},0}, e_0), (s_{\mathcal{V}_{loc},U}, e_U))$ iff there exist states $(s_{\mathcal{V}_{loc}}, s_{\mathcal{V}}), (t_{\mathcal{V}_{loc}}, t_{\mathcal{V}}) \in \mathcal{S}$ with $s_{\mathcal{V}_{loc},0} \subseteq s_{\mathcal{V}_{loc}}, s_{\mathcal{V}_{loc},U} \subseteq t_{\mathcal{V}_{loc}}$, and $e_0(s_{\mathcal{V}})$ and $e_U(t_{\mathcal{V}})$ evaluate to true, such that $(t_{\mathcal{V}_{loc}}, t_{\mathcal{V}})$ is reachable from $(s_{\mathcal{V}_{loc}}, s_{\mathcal{V}})$ in Θ^π. Otherwise Θ^π is *safe* with respect to $((s_{\mathcal{V}_{loc},0}, e_0), (s_{\mathcal{V}_{loc},U}, e_U))$.

In the definition above, the start condition characterizes $S_0 = \{(s_{\mathcal{V}_{loc}}, s_{\mathcal{V}}) \in \mathcal{S}_{\mathcal{V}_{loc}} \times \mathcal{S}_{\mathcal{V}} \mid s_{\mathcal{V}_{loc},0} \subseteq s_{\mathcal{V}_{loc}} \wedge e_0(s_{\mathcal{V}})\}$, while the unsafety condition characterizes $S_U = \{(s_{\mathcal{V}_{loc}}, s_{\mathcal{V}}) \in \mathcal{S}_{\mathcal{V}_{loc}} \times \mathcal{S}_{\mathcal{V}} \mid s_{\mathcal{V}_{loc},U} \subseteq s_{\mathcal{V}_{loc}} \wedge e_U(s_{\mathcal{V}})\}$.

Policy Predicate Abstraction

We use *predicate abstraction* (Graf and Saïdi 1997) as a method to analyze policy behavior. Concretely, we consider the predicate abstraction of the policy restricted state space, which we introduce as *policy predicate abstraction*. Generally speaking, predicates are logical formulas that evaluate to true or false over the states of a system. Given a set of predicates, a predicate abstraction is then obtained by mapping *concrete* states to *abstract* states according to the truth values of the predicates in the concrete state, while preserving transitions. Thus, each path possible in the concrete state space has a correspondence in the abstract state space, i.e., the predicate abstraction is an *over-approximation*.

We assume here for simplicity that the set of predicates is given. Future work will need to investigate the automatic generation of predicate sets e.g. via counter example guided abstraction refinement (Podelski and Rybalchenko 2007; Smaus and Hoffmann 2008). Note that the adaptation of such methods to our setting is quite non-trivial (the predicates ideally should distinguish regions of states due to different neural network behavior).

Predicate Abstraction for Networks of Automata For simplicity, in our concrete setting of networks of automata, we treat all locations explicitly and consider predicates from Exp_{bool}. More concretely, to guarantee that the negation of predicates is in Exp_{bool} again, we even restrict predicates to be linear integer inequations, i.e., constraints of the form $d_1 \cdot v_1 + \cdots + d_r \cdot v_r \lesseqqgtr c$ with negation $d_1 \cdot v_1 + \cdots + d_r \cdot v_r \gtreqqless c \pm 1$. Given a set $\mathcal{P} = \{p_1, \ldots, p_m\}$ of predicates from Exp_{bool}, an abstract state is then a tuple $(s_{\mathcal{V}_{loc}}, s_{\mathcal{P}})$, where $s_{\mathcal{V}_{loc}} \in \mathcal{S}_{\mathcal{V}_{loc}}$ is a complete location variable assignment and $s_{\mathcal{P}} \colon \mathcal{P} \to \mathbb{B}$ is a truth-value assignment complete over $\mathcal{P}$. We also refer to $s_{\mathcal{P}}$ as *predicate state*. The abstraction of a state variable assignment $s_{\mathcal{V}} \in \mathcal{S}_{\mathcal{V}}$ is the predicate state $s_{\mathcal{V}}|_{\mathcal{P}}$ with $s_{\mathcal{V}}|_{\mathcal{P}}(p) = p(s_{\mathcal{V}})$ for each $p \in \mathcal{P}$. Conversely, $[s_{\mathcal{P}}] = \{s_{\mathcal{V}} \in \mathcal{S}_{\mathcal{V}} \mid s_{\mathcal{V}}|_{\mathcal{P}} = s_{\mathcal{P}}\}$ denotes the *concretization* of predicate state $s_{\mathcal{P}}$.

Given the specifications above, we define (policy) predicate abstraction for automata networks as follows:

Definition 2 ((Policy) Predicate Abstraction)**.** Given the state space $\Theta = \langle \mathcal{S}, \mathcal{A}, \mathcal{T} \rangle$ of a network of automata and a predicate set $\mathcal{P} = \{p_1, \dots, p_m\}$ of inequations from Exp_{bool}, the *predicate abstraction* of Θ over $\mathcal{P}$ is the LTS $\Theta|_\mathcal{P} = \langle \mathcal{S}|_\mathcal{P}, \mathcal{A}, \mathcal{T}|_\mathcal{P} \rangle$, where $\mathcal{S}|_\mathcal{P} = \mathcal{S}_{\mathcal{V}_{loc}} \times (\mathcal{P} \to \mathbb{B})$, and $\mathcal{T}|_\mathcal{P} = \{((s_{\mathcal{V}_{loc}}, s_\mathcal{P}), a, (s'_{\mathcal{V}_{loc}}, s'_\mathcal{P})) \in \mathcal{S}|_\mathcal{P} \times \mathcal{A} \times \mathcal{S}|_\mathcal{P} \mid \exists (s_{\mathcal{V}_{loc}}, s_\mathcal{V}), (s'_{\mathcal{V}_{loc}}, s'_\mathcal{V}) \in \mathcal{S}\colon s_\mathcal{V} \in [s_\mathcal{P}], s'_\mathcal{V} \in [s'_\mathcal{P}] \wedge ((s_{\mathcal{V}_{loc}}, s_\mathcal{V}), a, (s'_{\mathcal{V}_{loc}}, s'_\mathcal{V})) \in \mathcal{T}\}$.

Given an action policy $\pi\colon \mathcal{S} \to \mathcal{A}$ for Θ, the *policy predicate abstraction* of Θ over $\mathcal{P}$ and π is the predicate abstraction of Θ^π over $\mathcal{P}$, i.e., $\Theta^\pi|_\mathcal{P} = \langle \mathcal{S}|_\mathcal{P}, \mathcal{A}, \mathcal{T}^\pi|_\mathcal{P} \rangle$.

Note that, $\mathcal{T}^\pi|_\mathcal{P} \subseteq \mathcal{T}|_\mathcal{P}$, since $\mathcal{T}^\pi \subseteq \mathcal{T}$. Hence, $\Theta|_\mathcal{P}$ can be utilized as an over-approximation of $\Theta^\pi|_\mathcal{P}$, from which one can improve by soundly removing transitions.

Policy Safety in $\Theta^\pi|_\mathcal{P}$ Given a policy safety property, the over-approximation property of predicate abstraction allows us to verify safety of Θ^π via reachability analyses in $\Theta^\pi|_\mathcal{P}$. Concretely, if Θ^π is unsafe, i.e., there is a path from a start state s to an unsafe state t, then there is a path from $s|_\mathcal{P}$ to $t|_\mathcal{P}$. This intuition is captured in the following proposition:

Proposition 3 (Policy Safety in $\Theta^\pi|_\mathcal{P}$)**.** If Θ^π is unsafe with respect to policy safety property $((s_{\mathcal{V}_{loc},0}, e_0), (s_{\mathcal{V}_{loc},U}, e_U))$, then there exist abstract states $(s_{\mathcal{V}_{loc}}, s_\mathcal{P}), (t_{\mathcal{V}_{loc}}, t_\mathcal{P}) \in \mathcal{S}|_\mathcal{P}$ where $s_{\mathcal{V}_{loc},0} \subseteq s_{\mathcal{V}_{loc}}, s_{\mathcal{V}_{loc},U} \subseteq t_{\mathcal{V}_{loc}}$ and there exist states $s_\mathcal{V} \in [s_\mathcal{P}], t_\mathcal{V} \in [t_\mathcal{P}]$ such that $e_0(s_\mathcal{V})$ and $e_U(t_\mathcal{V})$ hold, such that $(t_{\mathcal{V}_{loc}}, t_\mathcal{P})$ is reachable from $(s_{\mathcal{V}_{loc}}, s_\mathcal{P})$ in $\Theta^\pi|_\mathcal{P}$.

By contraposition, if there exists no path from an *abstract start state* to an *abstract unsafe state*, i.e., abstract states whose concretizations intersect with S_0 respectively S_U, then there also does not exists a path from a concrete start state to an concrete unsafe state. In other words, if $\Theta^\pi|_\mathcal{P}$ is *safe*, then so is Θ^π. Conversely, an unsafe $\Theta^\pi|_\mathcal{P}$ does **not** imply that Θ^π is unsafe too, since predicate abstraction may introduce spurious paths, i.e., paths without correspondence in the concrete state space.

SMT-Tests to Compute $\Theta^\pi|_\mathcal{P}$

The question whether there is a transition in $\Theta^\pi|_\mathcal{P}$ involves a satisfiability problem over state variable assignments. Given abstract states $(s_{\mathcal{V}_{loc}}, s_\mathcal{P}), (s'_{\mathcal{V}_{loc}}, s'_\mathcal{P}) \in \mathcal{S}|_\mathcal{P}$ and action $a = (g_{loc}, g, u_{loc}, u)$, with $g_{loc} \subseteq s_{\mathcal{V}_{loc}}$ and $s'_{\mathcal{V}_{loc}} = s_{\mathcal{V}_{loc}}[u_{loc}]$, i.e., the location constraints are fulfilled, then $((s_{\mathcal{V}_{loc}}, s_\mathcal{P}), a, (s'_{\mathcal{V}_{loc}}, s'_\mathcal{P})) \in \mathcal{T}^\pi|_\mathcal{P}$ iff there exist $s_\mathcal{V} \in [s_\mathcal{P}], s'_\mathcal{V} \in [s'_\mathcal{P}]$ such that $((s_{\mathcal{V}_{loc}}, s_\mathcal{V}), a, (s'_{\mathcal{V}_{loc}}, s'_\mathcal{V})) \in \mathcal{T}^\pi$.

Due to the underlying NN structure we refer to this as *NN-satisfiability problem*. Formally, such satisfiability problems can be encoded as tests in the context of satisfiability modulo theories (SMT) (Barrett et al. 1994) (in our case standard theories). We refer to these *SMT-tests* as *NN-SAT* tests.

Our algorithm to compute $\Theta^\pi|_\mathcal{P}$ applies various SMT-tests in general as well as NN-SAT tests in particular. In this section, we provide an overview of the SMT-tests. Details will be given in the technical report. In the next section, we present the actual algorithm. More concretely, while an SMT-test is the problem of deciding the satisfiability of a given formula, we focus on the abstract specification of SMT-tests, i.e., which conditions are checked via which test. An *SMT-encoding* is then any formula that is satisfiable iff the condition is fulfilled.

Transition Tests In the following, we give the specification for the NN-SAT-test of the transition condition in $\mathcal{T}^\pi|_\mathcal{P}$. Below, we also specify tests for partial conditions and relaxations. Since, typically, NN-SAT tests are computationally expensive, we additionally consider SMT-tests on transition conditions in $\mathcal{T}|_\mathcal{P}$, i.e., the transitions of the *non-policy-restricted* predicate abstraction, as over-approximations.

Definition 4 (Transition Tests of $\mathcal{T}|_\mathcal{P}$ and $\mathcal{T}^\pi|_\mathcal{P}$)**.** An SMT *transition test* of $\mathcal{T}|_\mathcal{P}$, denoted $SMT(s_\mathcal{P}, a, s'_\mathcal{P})$, tests the condition $\exists s_\mathcal{V} \in [s_\mathcal{P}], s'_\mathcal{V} \in [s'_\mathcal{P}]\colon g(s_\mathcal{V}) \wedge s'_\mathcal{V} = s_\mathcal{V}[u(s_\mathcal{V})]$.

An NN-SAT transition test of $\mathcal{T}^\pi|_\mathcal{P}$, denoted $NNSat(s_\mathcal{P}, a, s'_\mathcal{P})$, tests the condition $\exists s_\mathcal{V} \in [s_\mathcal{P}], s'_\mathcal{V} \in [s'_\mathcal{P}]\colon g(s_\mathcal{V}) \wedge s'_\mathcal{V} = s_\mathcal{V}[u(s_\mathcal{V})] \wedge \pi((s_{\mathcal{V}_{loc}}, s_\mathcal{V})) = a$.

While the encoding of $SMT(s_\mathcal{P}, a, s'_\mathcal{P})$ is standard to predicate abstraction approaches, $NNSat(s_\mathcal{P}, a, s'_\mathcal{P})$ extends this by an encoding of the policy condition $\pi((s_{\mathcal{V}_{loc}}, s_\mathcal{V})) = a$, especially the NN structure. In the case of a feed-forward NN, there is, for each neuron, a linear constraint over the inputs and a constraint over the piecewise-linear cases of the output. Indeed, for the NN analysis methods, that we query, it is crucial that the encoding is a conjunction of linear constraints and constraints for piecewise-linear activation functions.

Applicability Tests Besides the transition tests specified above, our algorithm also uses SMT-tests on partial transition conditions, more concretely, the applicability condition, i.e., whether for the abstract source state $(s_{\mathcal{V}_{loc}}, s_\mathcal{P})$ there exists at least one transition under a. If not, one can rule out the entire family of transition tests with source $(s_{\mathcal{V}_{loc}}, s_\mathcal{P})$ and action a at once. In the context of automata networks the applicability condition involves the satisfiability problem whether there exists $s_\mathcal{V} \in [s_\mathcal{P}]$ such that $g(s_\mathcal{V})$ is true (for $\mathcal{T}|_\mathcal{P}$) respectively whether simultaneously $\pi((s_{\mathcal{V}_{loc}}, s_\mathcal{V})) = a$ (for $\mathcal{T}^\pi|_\mathcal{P}$).

Definition 5 (Applicability Tests of $\mathcal{T}|_\mathcal{P}$ and $\mathcal{T}^\pi|_\mathcal{P}$)**.** An SMT *applicability test* of $\mathcal{T}|_\mathcal{P}$, denoted $SMT(s_\mathcal{P}, g)$, tests the condition $\exists s_\mathcal{V} \in [s_\mathcal{P}]\colon g(s_\mathcal{V})$.

An NN-SAT applicability test of $\mathcal{T}^\pi|_\mathcal{P}$, denoted $NNSat(s_\mathcal{P}, g, a)$, tests the condition $\exists s_\mathcal{V} \in [s_\mathcal{P}]\colon g(s_\mathcal{V}) \wedge \pi((s_{\mathcal{V}_{loc}}, s_\mathcal{V})) = a$.

An applicability tests can be encoded as a subformula of the respective transition test. Also, like for the transition tests, $SMT(s_\mathcal{P}, g)$ over-approximates $NNSat(s_\mathcal{P}, g, a)$.

Relaxed NN-SAT Tests As mentioned above, we use tests for $\mathcal{T}|_\mathcal{P}$ to over-approximate more expensive NN-SAT tests for $\mathcal{T}^\pi|_\mathcal{P}$. Another option to over-approximate is via *relaxed* NN-SAT tests, i.e., tests that relax the discrete integer state variables to the continuous real domain.

Definition 6 (Relaxed NN-SAT tests)**.** The continuously-relaxed version of an NN-SAT test $NNSat(s_\mathcal{P}, a, s'_\mathcal{P})$ is denoted $NNSat_\mathbb{R}(s_\mathcal{P}, a, s'_\mathcal{P})$.

The continuously-relaxed version of an NN-SAT test $NNSat(s_\mathcal{P}, g, a)$ is denoted $NNSat_\mathbb{R}(s_\mathcal{P}, g, a)$.

Clearly, this relaxation is sound, i.e., if the relaxed NN-SAT test is not fulfilled, then the *exact* NN-SAT test on the discrete domain is not fulfilled too. The motivation is that, on the continuous domain we can query efficient off-the-shelf NN analysis methods, in particular NN-SAT solvers (Katz et al. 2017). Moreover, the precision of the obtained over-approximation may actually suffice the purpose, i.e., safety verification.

Algorithm 1: Exact NN-SAT via Branch & Bound.

```
1  Procedure nn_sat_bb (s_P, a, s'_P):
2    if ¬NNSat_R(s_P, a, s'_P) then
3      return false
4    else
5      s_V ← NNSat_R(s_P, a, s'_P)
6      for each v ∈ V do
7        if ¬is_integer(s_V(v)) then
8          let lo(v) ← ceil(s_V(v)) in
9            if nn_sat_bb (s_P, a, s'_P) then
               return true;
10         let up(v) ← floor(s_V(v)) in
11           return nn_sat_bb (s_P, a, s'_P)
12   return true
```

Exact NN-SAT via Branch & Bound Given a solver for relaxed NN-SAT tests, we can implement a sound and complete decision procedure for exact NN-SAT tests querying the relaxed solver as an oracle in a *branch & bound* procedure (Little et al. 1963). The method is illustrated for $NNSat(s_{\mathcal{P}}, a, s'_{\mathcal{P}})$ in Algorithm 1. If $NNSat_{\mathbb{R}}(s_{\mathcal{P}}, a, s'_{\mathcal{P}})$ is not fulfilled (line 2), then so is $NNSat(s_{\mathcal{P}}, a, s'_{\mathcal{P}})$, and we can terminate. Otherwise, we extract the underlying solution for the source state $s_{\mathcal{V}}$. Note that, the target state $s'_{\mathcal{V}}$ is integer if $s_{\mathcal{V}}$ is, and does not need to be considered explicitly. We check for each state variable $v \in \mathcal{V}$ whether $s_{\mathcal{V}}(v)$ is integer (line 7). If so, the exact test is fulfilled and we can terminate (line 12). If not, we branch on v, once bounding v to the (integer) subdomain strictly larger than $s_{\mathcal{V}}(v)$ (line 8), and once to the subdomain strictly smaller than $s_{\mathcal{V}}(v)$ (line 10). Analogously to other branch & bound frameworks, one can prove that eventually in at least one of the branches an integer solution is found iff $NNSat(s_{\mathcal{P}}, a, s'_{\mathcal{P}})$ is fulfilled.

Membership Tests Towards safety verification, we also need to decide membership in the set of abstract start states $S_0|_{\mathcal{P}}$, respectively the set of abstract unsafe states $S_U|_{\mathcal{P}}$. Given a policy safety property $((s_{\mathcal{V}_{loc},0}, e_0), (s_{\mathcal{V}_{loc},U}, e_U))$ and an abstract state $(s_{\mathcal{V}_{loc}}, s_{\mathcal{P}}) \in \mathcal{S}|_{\mathcal{P}}$ with $s_{\mathcal{V}_{loc},0} \subseteq s_{\mathcal{V}_{loc}}$, respectively $s_{\mathcal{V}_{loc},U} \subseteq s_{\mathcal{V}_{loc}}$, the membership condition comes down to an SMT-test:

Definition 7 (Membership Tests)**.** An SMT *membership test* of $S_0|_{\mathcal{P}}$ checks the condition $\exists s_{\mathcal{V}} \in [s_{\mathcal{P}}] \colon e_0(s_{\mathcal{V}})$.

An SMT membership test of $S_U|_{\mathcal{P}}$ checks the condition $\exists s_{\mathcal{V}} \in [s_{\mathcal{P}}] \colon e_U(s_{\mathcal{V}})$.

Computing $\Theta^\pi|_{\mathcal{P}}$ Starting from $S_0|_{\mathcal{P}}$

In this section, we outline the the computation of $\Theta^\pi|_{\mathcal{P}}$ for a given NN policy π and a set of predicates $\mathcal{P}$. More concretely, since we are interested in safety verification for a given policy safety property $((s_{\mathcal{V}_{loc},0}, e_0), (s_{\mathcal{V}_{loc},U}, e_U))$, we compute for the set of abstract start states $S_0|_{\mathcal{P}}$ the **reachable fragment** of $\Theta^\pi|_{\mathcal{P}}$, i.e., an over-approximation of the fragment of Θ^π reachable from S_0. While our approach applies the SMT-tests presented above, it is completely modular in the concrete methods queried to answer these tests.

In what follows, we present an algorithm for state expansion, i.e., given an abstract state $(s_{\mathcal{V}_{loc}}, s_{\mathcal{P}}) \in \mathcal{S}|_{\mathcal{P}}$ and an action $a \in \mathcal{A}$ to compute all outgoing transitions $((s_{\mathcal{V}_{loc}}, s_{\mathcal{P}}), a, (s'_{\mathcal{V}_{loc}}, s'_{\mathcal{P}})) \in \mathcal{T}^\pi|_{\mathcal{P}}$. Given this algorithm, we compute the reachable fragment of $\Theta^\pi|_{\mathcal{P}}$ in a forward search from $S_0|_{\mathcal{P}}$. Θ^π is safe, if $(s'_{\mathcal{V}_{loc}}, s'_{\mathcal{P}}) \notin S_U|_{\mathcal{P}}$ for each reached abstract state.

State Expansion Algorithm 2 shows the state expansion in $\Theta^\pi|_{\mathcal{P}}$ for abstract source state $(s_{\mathcal{V}_{loc}}, s_{\mathcal{P}}) \in \mathcal{S}|_{\mathcal{P}}$ and action $a \in \mathcal{A}$. Up to optimizations the algorithm works as follows: If the source location constraint is fulfilled (line 1), we fix $s'_{\mathcal{V}_{loc}}$ according to the location update u_{loc} (line 5). Subsequently, we enumerate **each** potential successor predicate state $s'_{\mathcal{P}}$ (line 8) and check via the NN-SAT transition test whether a transition to $(s'_{\mathcal{V}_{loc}}, s'_{\mathcal{P}})$ exists (line 13).

Algorithm 2: State Expansion.

```
   Input: (s_Vloc, s_P) ∈ S|_P, a ∈ A with
          a = (g_loc, g, u_loc, u)
1  if ¬g_loc ⊆ s_Vloc then return
   // optional applicability tests:
2  if ¬SMT(s_P, g) then return
3  if ¬NNSat_R(s_P, g, a) then return
4  if ¬NNSat(s_P, g, a) then return
5  s'_Vloc := s_Vloc[u_loc]
6  let s'_P ∈ P → B with dom(s'_P) = ∅ in
7    s'_P fixed with respect to s_P, g, u          // opt
8    enumerate_states(s'_P)
9  Procedure enumerate_states(s'_P: predicate state):
10   if dom(s'_P) = P then
       // optional transition tests:
11     if ¬SMT(s_P, a, s'_P) then return
12     if ¬NNSat_R(s_P, a, s'_P) then return
13     if NNSat(s_P, a, s'_P) then
14       add ((s_Vloc, s_P), a, (s'_Vloc, s'_P)) to T^π|_P
15   else
16     for some p ∈ P \ dom(s'_P)
17     let s'_P(p) ← true in
18       s'_P fixed with respect to p              // opt
19       enumerate_states(s'_P)
20     let s'_P(p) ← false in
21       s'_P fixed with respect to p              // opt
22       enumerate_states(s'_P)
```

Optimization & Over-Approximation In Algorithm 2, we show a number of optional tests, that are sound, i.e., they are fulfilled for transitions in $\mathcal{T}^{\pi}|_{\mathcal{P}}$. The motivation is that SMT-tests, especially NN-SAT tests, are computationally expensive. Hence, in practice, reducing the number of SMT-tests, especially NN-SAT tests, is desirable.

The number of enumerated predicate states (line 8), and thus the number of transition tests to apply, is exponential in the number of predicates m. Therefore, we aim at reducing the number of state expansions for which enumerate via applicability tests (line 2 et sqq.). Furthermore, we reduce the number of exact NN-SAT transition tests via over-approximating transition tests (line 11 et sqq.).

While the specified ordering applies typically less expensive and/or less precise tests first, each test may or may not be applied independently of others. In fact, dropping the NN-SAT transition test (line 13), Algorithm 2 can also compute over-approximations of $\Theta^{\pi}|_{\mathcal{P}}$. That said, our approach is highly modular and extensible in both, the methods it queries to answer SMT-tests but also the tests it applies.

Predicate State Enumeration The predicate state enumeration exponential in m poses a major difficulty towards the feasibility of the approach. Hence, besides action applicability tests, we also try to reduce the number of enumerated states themselves.

In Algorithm 2, the procedure `enumerate_states` (line 9) is called with an empty predicate state $s'_{\mathcal{P}}$ (line 8) and sets predicate values recursively. At each inner node (line 16) of the recursion tree, one branches over a predicate, for which $s'_{\mathcal{P}}$ is not yet defined, resulting in one branch where the predicate is set true (line 17) and one where it is set false (line 20). At each leaf node (line 10) all predicates are set yielding a predicate state unique over all leaves. Given this enumeration base, we apply optimizations to prune the recursion tree and thereby reduce the number of enumerated states.

One optimization uses *binary predicate relations*: A truth value assignment to one predicate p may entail truth values of others predicates. We compute this information once prior to the search via SMT-tests. Then, if p is set, we *fix* further truth values in $s'_{\mathcal{P}}$ with respect to p (line 18, 21). Another option is to apply an optimization inspired by *Cartesian abstraction* (Ball, Podelski, and Rajamani 2003): Ahead of each state enumeration (line 7), we apply SMT-tests to check for each predicate individually whether a certain truth value is entailed given $s_{\mathcal{P}}, g$ and u. For both optimizations, each fixed predicate reduces the recursion tree below the fix by a factor of 2.

Experiments

We have implemented our approach on top of a C++ code base for automata networks modeled in JANI (Budde et al. 2017). All experiments were run on machines with Intel Xenon E5-2650 processors with a clock rate of 2.2 GHz with time and memory limits of 12 h and 4 GB respectively.

Our evaluation is exclusively focused on *scalability*, as a core question in an approach that ultimately tackles the state space explosion compounded by NN-satisfiability tests. We do not consider the actual results of the verification process, i.e., whether and where π is safe. In particular, we build the entire part of $\Theta|_{\mathcal{P}}$ reachable from $S_0|_{\mathcal{P}}$, continuing even if we already reached an abstract unsafe state.

Evaluated Methods We consider three concrete methods to compute the reachable fragment of $\Theta^{\pi}|_{\mathcal{P}}$ (PA+BB(Marabou), PA+Marabou+Z3) respectively over-approximations thereof (PA+Marabou). In accordance with our specifications above, all methods do so in a sequential forward search from the set of abstract start states and with a state expansion as per Algorithm 2. The underlying predicate abstraction base (PA) queries the $Z3$ solver (de Moura and Bjorner 2008) for SMT-tests (Alg. 2 line 2, 11). PA+Marabou improves from this over-approximation base by applying relaxed NN-SAT tests (line 3, 11) implemented by *Marabou* (Katz et al. 2019). PA+BB(Marabou) extends this method by querying *Marabou* in branch & bound to compute the exact NN-SAT transition tests (line 13). Conversely, PA+Marabou+Z3 extends PA+Marabou by querying $Z3$ for the exact NN-SAT transition test. Here, to improve the feasibility, we also feed neuron value bounds derived by *Marabou* into the $Z3$ queries.

Racetrack In our experiments we use a version of Racetrack (Gros et al. 2020) modeled in JANI. In short, a car must drive on a discretized 2-dimensional map. It crashes when it drives into a wall or off the map. The policy task is to control the car's acceleration via 9 different actions. The car is described by two position and two velocity components.

In the original JANI model accelerations may fail with a set probability. Since we consider a non-probabilistic setting, we set this probability to 0. Furthermore, to focus on the impact of NN analysis, we simplify the model, more concretely, the collision detection: For each discrete position and velocity update after an acceleration, it is only checked whether the new position is a wall or off-map, but not whether the car hit a wall on the corresponding path from old to new position. In all our experiments, we consider the map *Barto-small* (Gros et al. 2020) with dimensions 35×12.

NN Policies We train policies using the infrastructure of (Gros et al. 2020), i.e., an application of Q-learning (Mnih et al. 2015). However, to keep the NN input encodings simple, we restrict the input features to be the four position and velocity components. The underlying learning objective is to reach certain goal positions while avoiding to crash. We train four NNs in total, each with two hidden layers but varying in the number of neurons per layer, namely $16, 24, 32, 64$.

Safety Property In accordance with the policy learning, the underlying safety property we consider is the one of *not crashing*. However, we want to focus on analyzing the scalability of the tested methods rather than actually verifying policies. In terms of this scalability analysis, the set of (reachable) unsafe states is of secondary interest to us. In contrast, since we start the forward search from the set of abstract start states, the set of start states is certainly not. Besides a start state with velocity 0 already provided by

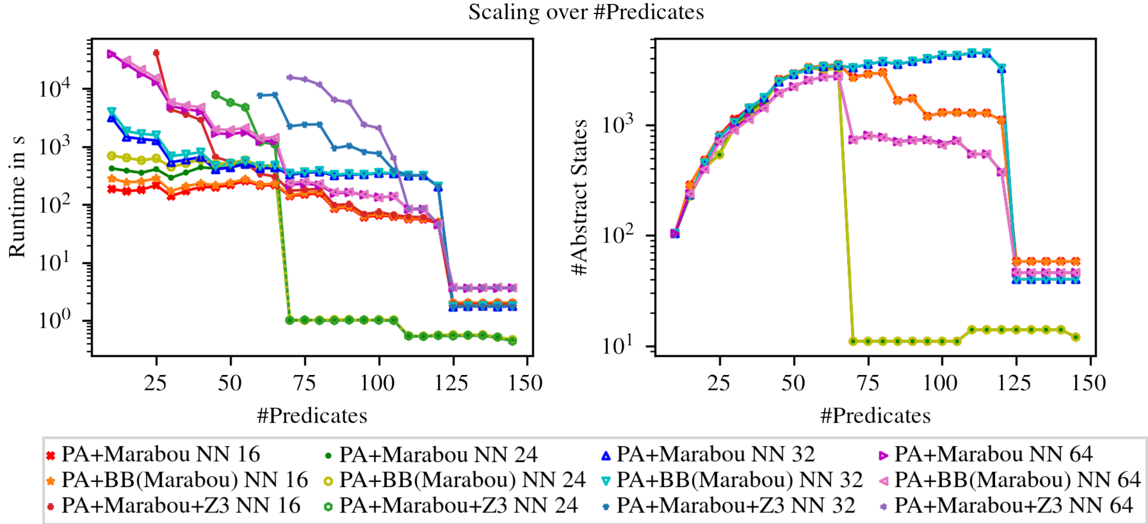

Figure 1: Time to compute the abstract state space (left) and the number of reachable abstract states (right) dependent on the size of the predicate set, for a single start state, different methods to compute $\Theta^\pi|_\mathcal{P}$ (PA+BB(Marabou), PA+Marabou+Z3) respectively over-approximations (PA+Marabou) thereof, and NN policies with 2 hidden layers of size $16, 24, 32$ and 64.

the racetrack model, we generate 1000 **random start states** with varying position and velocity components. All those states are solvable, i.e., the car may reach a goal position under some policy. In our evaluation we consider start sets of varying size that are subsets of the generated start states.

Predicate Sets We consider predicates of the form $c \leq v$, where c is an integer constant and v is a position or velocity variable. We choose the predicate sets manually and use a fixed procedure determining the predicates added for each v: The first predicate splits the domain into two halves. The remaining number of predicates to be added is then evenly distributed on the resulting subdomains, on which we proceed recursively. Therewith, for each variable, predicates are added in a fixed order.

Scalability over $|\mathcal{P}|$ In our first analysis, we consider a single concrete start state and predicate sets of varying size. This setup is of course not relevant in practice (one could simply run the deterministic policy from the unique start state), but it nevertheless makes sense as a scalability study of abstract state space construction: to evaluate how our methods scale as a function of abstraction granularity and NN size, and to investigate the precision of the over-approximation computed by PA+Marabou. We start with a single predicate for each of the position and velocity variables, and then stepwise add one predicate for one variable.

Figure 1 shows the results in terms of the time to compute (left) as well as size of (right) the reachable abstract state space. Concerning the latter, we only include results for one exact method, namely PA+BB(Marabou) (but not PA+Marabou+Z3). We run an experiment for every 5th predicate set according to the scheme specified above. A less compact illustration can be found in the appendices.

A somewhat surprising observation is, that PA+Marabou computes, also for coarse predicate sets, a rather fine over-approximation – with but only with the precision of the plot indeed $\Theta^\pi|_\mathcal{P}$ itself. Further manual analysis shows that there are cases where the relaxed NN-SAT transition test is fulfilled but the transition is pruned due to the exact SMT transition test of PA. In other words, the precision of PA+Marabou may be due to an interplay of the precision gained by relaxed NN-SAT tests combined with exact SMT tests. That said, experiments on other domains will be needed to further evaluate the precision gained via PA+Marabou.

Given the precision of PA+Marabou, a rather expected result is, that over all instances PA+Marabou dominates PA+BB(Marabou) in terms of runtime. Moreover, PA+BB(Marabou) dominates PA+Marabou+Z3, since, in contrast to $Z3$, *Marabou* is highly specialized to NN analysis. For finer $\mathcal{P}$ the runtime difference diminishes as the precision of relaxed tests increases.

Observe that the runtime correlates with the size of the abstract state space only for finer predicate sets. For coarser $\mathcal{P}$ the runtime is dominated by the relaxed NN-SAT test. More precisely, while the number of relaxed NN-SAT tests is linear in the number of states, the time needed to solve a query over an NN is highly dependent on the size of the NN's input region which in turn depends on the granularity of $\mathcal{P}$. Thus, for coarse predicate sets the average time to solve an relaxed NN-SAT test, and therewith the overall computation time, is increased. On the other hand, as one would expect, over different NN the runtime grows with the size of the NN.

Finally, one can observe that as $\mathcal{P}$ becomes larger the runtime and the size of the abstract state space eventually drop. Initially, the size of reachable $\Theta^\pi|_\mathcal{P}$ grows with the general abstract state space $\Theta|_\mathcal{P}$ for finer $\mathcal{P}$. However, the finer $\mathcal{P}$,

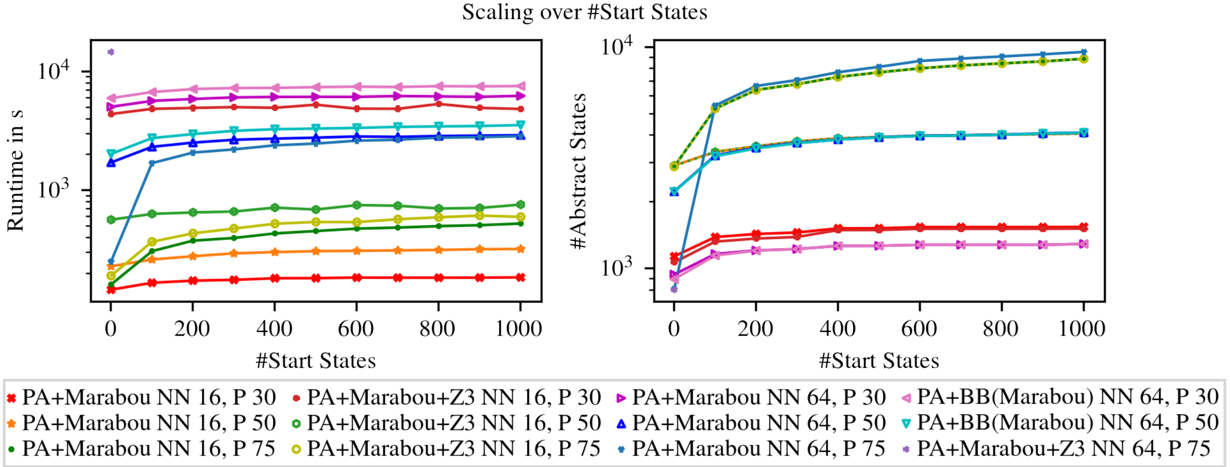

Figure 2: Time to compute the abstract state space (left) and the number of reachable abstract states (right) dependent on the number of concrete start states, for different methods (PA+Marabou, PA+BB(Marabou), PA+Marabou+Z3), NN policies with 2 hidden layers of size 16 and 64, and for predicate sets of different size $(30, 50, 75)$.

the better the small reachable fragment of Θ^π is approximated. Here, the dropping point(s) depend on an interplay of $\mathcal{P}$ and π.

Scalability over $|S_0|$ We now evaluate the realistic setup where the number of start states $|S_0|$ scales. We do so for a number of settings of the other parameters (number of predicates and NN size); complete results are moved to the appendices. Starting with a single start set, we stepwise add 100 states from the set of random start states. Figure 2 shows the results. To preserve visibility, for each predicate set, we include either PA+BB(Marabou) or PA+Marabou+Z3. If PA+BB(Marabou) is included, then PA+Marabou+Z3 did not terminate on any instance. If PA+Marabou+Z3 is included, then PA+BB(Marabou)'s runtime equals the one of PA+Marabou with the precision of the plot.

Like in the first analysis, we see that PA+Marabou computes a rather fine over-approximation of $\Theta^\pi|_\mathcal{P}$. Indeed, with the precision of the plot, only for NN 16 and $|\mathcal{P}| = 30$ an information loss is visible. We again also see a significant but approximately constant runtime gap between PA+Marabou and the respective exact method, in particular for PA+Marabou+Z3.

Independent of the concrete method, the results also indicate the potential of predicate abstraction to tackle $|S_0|$ as a source of complexity towards policy safety verification. Concretely, after the first 100 random start states, the number of reachable states, and thereby the time to compute the state space, increases only slightly. However, further experiments, including actual policy safety verification, remain necessary to demonstrate the usefulness of this potential in practice.

Conclusion

We have introduced policy predicate abstraction as a technique to enable NN action policy analysis. We have designed an algorithm to compute policy predicate abstractions, that is highly modular and extensible with respect to the partial and/or relaxed transition conditions it tests and the efficient off-the-shelf NN analysis approaches it queries. The empirical results indicate that safety verification via policy predicate abstraction may be feasible, at least in a deterministic setting with the number of start states as the only source of complexity. Moreover, the results also show that rather fine, but significantly less expensive, over-approximations can be obtained via variations of our algorithm. These findings are preliminary however and more experiments will be needed to assess the approach with confidence.

Besides *Marabou*, there remains to consider a broad range of NN analysis approaches, in particular in the context of NN robustness verification, that can be queried to over-approximate transition conditions, e.g., *DeepSymbol* (Li et al. 2019). Moreover, one may investigate how those approaches may benefit from each other. For instance, the bounds on the neuron values derived by one method, e.g., *DeepSymbol*, may be provided to another, e.g., *Marabou*.

Alternatively, we also plan to build on adversarial attack methods to under-approximate transition conditions (Goodfellow, Shlens, and Szegedy 2014), as well as to leverage robustness guarantees on adversarial attacks (Hein and Andriushchenko 2017) to check the exact transition condition.

An important future task of course remains to investigate policy predicate abstraction with non-deterministic transitions and/or policy decisions. Overall, we believe that NN policy verification is important, and should be addressed not only in the formal methods community but also in the AI community itself. We hope that our work will provide one basic building block for this huge endeavor.

Acknowledgments

This work was funded by DFG Grant 389792660 as part of TRR 248 (CPEC, https://perspicuous-computing.science).

References

Alshiekh, M.; Bloem, R.; Ehlers, R.; Könighofer, B.; Niekum, S.; and Topcu, U. 2017. Safe Reinforcement Learning via Shielding. *CoRR* abs/1708.08611.

Ball, T.; Majumdar, R.; Millstein, T. D.; and Rajamani, S. K. 2001. Automatic predicate abstraction of C programs. In *22nd Conference on Programming Language Design and Implementation (PLDI)*.

Ball, T.; Podelski, A.; and Rajamani, S. K. 2003. Boolean and Cartesian abstraction for model checking C programs. *International Journal on Software Tools for Technology Transfer* 5(1): 49–58.

Barrett, C. W.; Sebastiani, R.; Seshia, S. A.; and Tinelli, C. 1994. Satisfiability modulo theories. *In Handbook of Satisfiability* 825–885.

Barto, A. G.; Bradtke, S. J.; and Singh, S. P. 1995. Learning to Act Using Real-Time Dynamic Programming. *Artif. Intell.* 72(1-2): 81–138.

Bonet, B.; and Geffner, H. 2003. Labeled RTDP: Improving the Convergence of Real-Time Dynamic Programming. In *ICAPS*, 12–21.

Budde, C. E.; Dehnert, C.; Hahn, E. M.; Hartmanns, A.; Junges, S.; and Turrini, A. 2017. JANI: Quantitative Model and Tool Interaction. In *TACAS (2)*, LNCS 10206, 151–168.

de Moura, L.; and Bjorner, N. 2008. Z3: An Efficient SMT Solver. In Ramakrishnan, C.; and Rehof, J., eds., *Tools and Algorithms for the Construction and Analysis of Systems. TACAS 2008*, LNCS 4963. Berlin, Heidelberg: Springer. https://doi.org/10.1007/978-3-540-78800-3_24.

Edelkamp, S. 2001. Planning with Pattern Databases. In *Proceedings of the 6th European Conference on Planning (ECP'01)*, 13–24.

Fulton, N.; and Platzer, A. 2018. Safe Reinforcement Learning via Formal Methods: Toward Safe Control Through Proof and Learning. In *Proc. 32nd AAAI Conference on Artificial Intelligence (AAAI'18)*.

Garg, S.; Bajpai, A.; and Mausam. 2019. Size Independent Neural Transfer for RDDL Planning. In *Proceedings of the 29th International Conference on Automated Planning and Scheduling (ICAPS 2019)*, 631–636.

Gehr, T.; Mirman, M.; Drachsler-Cohen, D.; Tsankov, P.; Chaudhuri, S.; and Vechev, M. T. 2018. AI2: Safety and Robustness Certification of Neural Networks with Abstract Interpretation. In *IEEE Symposium on Security and Privacy 2018*, 3–18.

Goodfellow, I. J.; Shlens, J.; and Szegedy, C. 2014. Explaining and Harnessing Adversarial Examples. doi:\url{https://arxiv.org/abs/1412.6572v1}.

Graf, S.; and Saïdi, H. 1997. Construction of abstract state graphs with PVS. In *9th International Conference on Computer Aided Verification (CAV)*.

Gros, T. P.; Hermanns, H.; Hoffmann, J.; Klauck, M.; and Steinmetz, M. 2020. Deep Statistical Model Checking. In *Proceedings of the 40th International Conference on Formal Techniques for Distributed Objects, Components, and Systems (FORTE'20)*. Available at https://doi.org/10.1007/978-3-030-50086-3_6.

Groshev, E.; Goldstein, M.; Tamar, A.; Srivastava, S.; and Abbeel, P. 2018. Learning Generalized Reactive Policies Using Deep Neural Networks. In *Proceedings of the 28th International Conference on Automated Planning and Scheduling (ICAPS'18)*, 408–416.

Hein, M.; and Andriushchenko, M. 2017. Formal Guarantees on the Robustness of a Classifier against Adversarial Manipulation. In Guyon, I.; von Luxburg, U.; Bengio, S.; Wallach, H. M.; Fergus, R.; Vishwanathan, S. V. N.; and Garnett, R., eds., *Proceedings of the 30th Annual Conference on Neural Information Processing Systems (NIPS)*, 2263–2273.

Helmert, M.; Haslum, P.; Hoffmann, J.; and Nissim, R. 2014. Merge & Shrink Abstraction: A Method for Generating Lower Bounds in Factored State Spaces. *Journal of the Association for Computing Machinery* 61(3): 16:1–16:63.

Henzinger, T. A.; Jhala, R.; Majumdar, R.; ; and McMillan, K. L. 2004. Abstractions from proofs. In *31st Symposium on Principles of Programming Languages (POPL)*.

Hoffmann, J.; Hermanns, H.; Klauck, M.; Steinmetz, M.; Karpas, E.; and Magazzeni, D. 2020. Let's Learn Their Language? A Case for Planning with Automata-Network Languages from Model Checking. In *Proceedings of the 34th AAAI Conference on Artificial Intelligence (AAAI'20)*, 13569–13575.

Holzmann, G. 2004. *The Spin Model Checker - Primer and Reference Manual*. Addison-Wesley.

Issakkimuthu, M.; Fern, A.; and Tadepalli, P. 2018. Training Deep Reactive Policies for Probabilistic Planning Problems. In de Weerdt, M.; Koenig, S.; Röger, G.; and Spaan, M. T. J., eds., *Proceedings of the 28th International Conference on Automated Planning and Scheduling (ICAPS)*, 422–430.

Katz, G.; Barrett, C. W.; Dill, D. L.; Julian, K.; and Kochenderfer, M. J. 2017. Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks. In *CAV (1)*, LNCS 10426, 97–117.

Katz, G.; Huang, D. A.; Ibeling, D.; Julian, K.; Lazarus, C.; Lim, R.; Shah, P.; Thakoor, S.; Wu, H.; Zeljic, A.; Dill, D. L.; Kochenderfer, M.; and Barrett, C. 2019. The Marabou Framework for Verification and Analysis of Deep Neural Networks. In Dillig, I.; and Tasiran, S., eds., *Computer Aided Verification. CAV 2019*, LNCS 11561. Cham: Springer. https://doi.org/10.1007/978-3-030-25540-4_26.

Könighofer, B.; Alshiekh, M.; Bloem, R.; Humphrey, L.; Könighofer, R.; Topcu, U.; and Wang, C. 2017. Shield synthesis. *Formal Methods in System Design* 51(2): 332–361.

Li, J.; Liu, J.; Yang, P.; Chen, L.; Huang, X.; and Zhang, L. 2019. Analyzing deep neural networks with symbolic propagation: Towards higher precision and faster verification. In Chang, B.-Y. E., ed., *Static Analysis – 26th International Symposium, SAS 2019*, LNCS 11822, 296–319. Porto, Portugal: Springer.

Little, J. D. C.; Murty, K. G.; Sweeney, D. W.; and Karel, C. 1963. An Algorithm for the Traveling Salesman Problem. Operations Research.

Mnih, V.; Kavukcuoglu, K.; Silver, D.; Graves, A.; Antonoglou, I.; Wierstra, D.; and Riedmiller, M. 2013. Playing Atari with Deep Reinforcement Learning. In *Proceedings of NIPS*.

Mnih, V.; Kavukcuoglu, K.; Silver, D.; Rusu, A. A.; Veness, J.; Bellemare, M. G.; Graves, A.; Riedmiller, M. A.; Fidjeland, A.; Ostrovski, G.; Petersen, S.; Beattie, C.; Sadik, A.; Antonoglou, I.; King, H.; Kumaran, D.; Wierstra, D.; Legg, S.; and Hassabis, D. 2015. Human-level control through deep reinforcement learning. *Nature* 518: 529–533.

Nair, V.; and Hinton, G. 2010. Rectified Linear Units Improve Restricted Boltzmann Machines. In *Proc. 27th Int. Conf. on Machine Learning (ICML)*, 807–814.

Podelski, A.; and Rybalchenko, A. 2007. ARMC: the logical choice for software model checking with abstraction refinement. In Hanus, M., ed., *Proceedings of the 9th International Symposium on Practical Aspects of Declarative Languages*, LNCS 4354, 245–259.

Sarle, W. 1994. Neural networks and statistical models.

Seipp, J.; and Helmert, M. 2018. Counterexample-Guided Cartesian Abstraction Refinement for Classical Planning. *Journal of Artificial Intelligence Research* 62: 535–577.

Silver, D.; Huang, A.; Maddison, C. J.; Guez, A.; Sifre, L.; van den Driessche, G.; Schrittwieser, J.; Antonoglou, I.; Panneershelvam, V.; Lanctot, M.; Dieleman, S.; Grewe, D.; Nham, J.; Kalchbrenner, N.; Sutskever, I.; Lillicrap, T.; Leach, M.; Kavukcuoglu, K.; Graepel, T.; and Hassabis, D. 2016. Mastering the Game of Go with Deep Neural Networks and Tree Search. *Nature* 529: 484–503.

Silver, D.; Hubert, T.; Schrittwieser, J.; Antonoglou, I.; Lai, M.; Guez, A.; Lanctot, M.; Sifre, L.; Kumaran, D.; Graepel, T.; Lillicrap, T.; Simonyan, K.; and Hassabis, D. 2018. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science* 362(6419): 1140–1144.

Smaus, J.; and Hoffmann, J. 2008. Relaxation Refinement: A New Method to Generate Heuristic Functions. In *Proceedings of the 5th International Workshop on Model Checking and Artificial Intelligence (MoChArt'08)*, Electronic Notes in Theoretical Computer Science.

Toyer, S.; Thiébaux, S.; Trevizan, F. W.; and Xie, L. 2020. ASNets: Deep Learning for Generalised Planning. *Journal of Artificial Intelligence Research* 68: 1–68.

Appendices

Further Illustrations of the Empirical Results In Figure 3, we distribute the runtime results from Figure 1 (scaling $|\mathcal{P}|$) over three plots for better visibility (top left, bottom left & right). We again include the plot on the number of reachable abstract states (top tight) from Figure 1 to allow direct comparison.

In Figure 4 we distribute the runtime results from Figure 2 (scaling $|S_0|$) to one plot per predicate set. We now also include results for $|\mathcal{P}| = 125$ and plot PA+BB(Marabou) and PA+Marabou+Z3 for each $\mathcal{P}$. Concerning the number of reachable abstract states, recall that the state space computed by PA+BB(Marabou) and PA+Marabou+Z3 is the same. Thus, we only print the number of states for one, namely the former, since it terminates on all instances. This especially allows to confirm the precision of PA+Marabou also for $|\mathcal{P}| = 75$. In Figure 2 we only had partial results since PA+Marabou+Z3 does not terminate for $|S_0| > 1$. For comparison, we now also print the number of abstract start states as a function of the concrete start states.

In Figure 5 we give results on PA+Marabou scaled over $|S_0|$ for the remaining neural networks (NN 24, NN 32). Here, we take a slightly different perspective and focus on the scaling results over different predicate sets (namely $|\mathcal{P}| = 30, 50, 75, 100, 125, 145$) rather than comparison to the exact methods. Similar to previous observations, we see that scaled over $|S_0|$, the size of the reachable fragment of $\Theta^\pi|_\mathcal{P}$ (respectively the computed over-approximation) initially grows for finer $\mathcal{P}$ with the general predicate abstraction $\Theta|_\mathcal{P}$, but eventually drops as reachable Θ^π is approximated more faithfully. This in particular applies to NN 24.

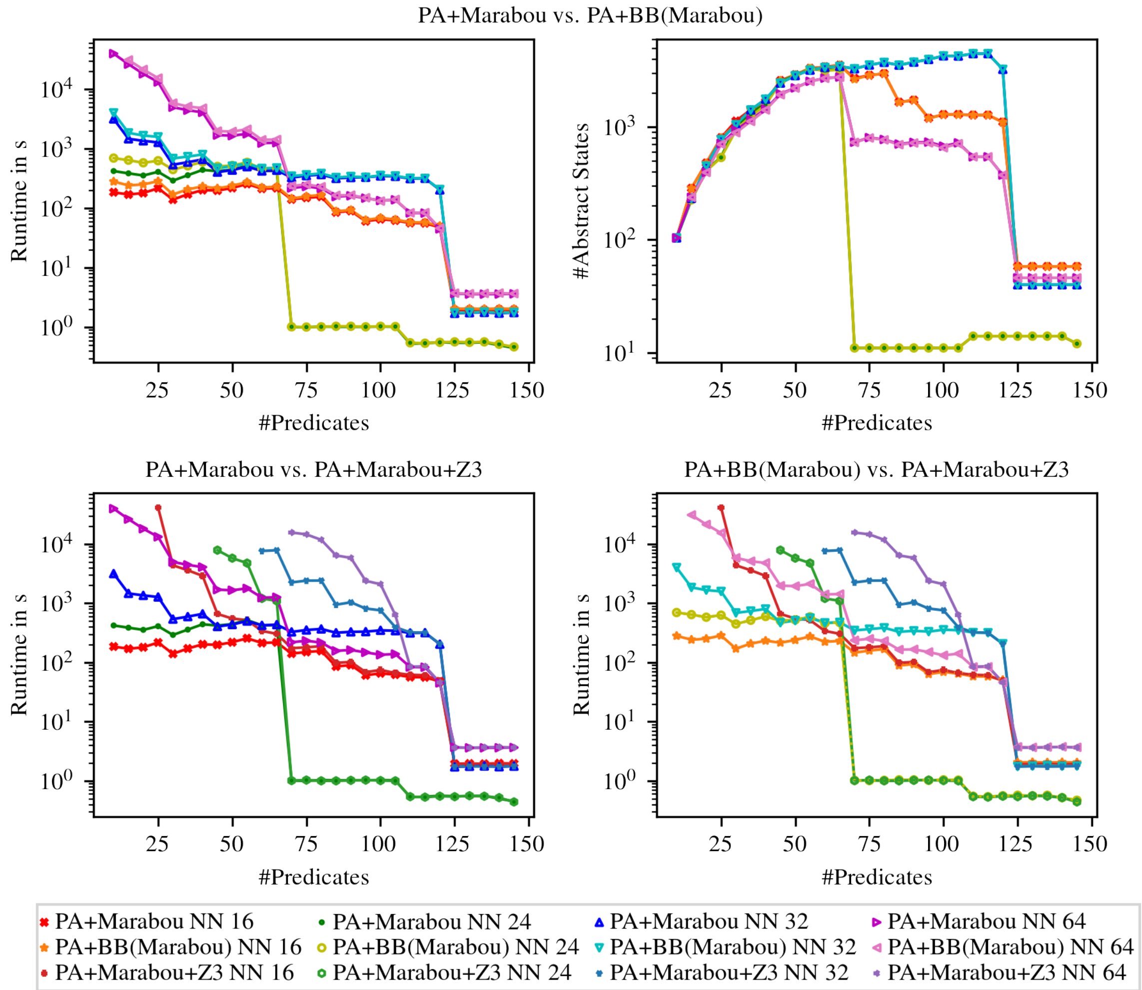

Figure 3: Time to compute the abstract state space and the number of reachable abstract states (top right) dependent on the size of the predicate set, for a single start state, different methods (PA+Marabou, PA+BB(Marabou), PA+Marabou+Z3) and NN policies with 2 hidden layers of size $16, 24, 32$ and 64.

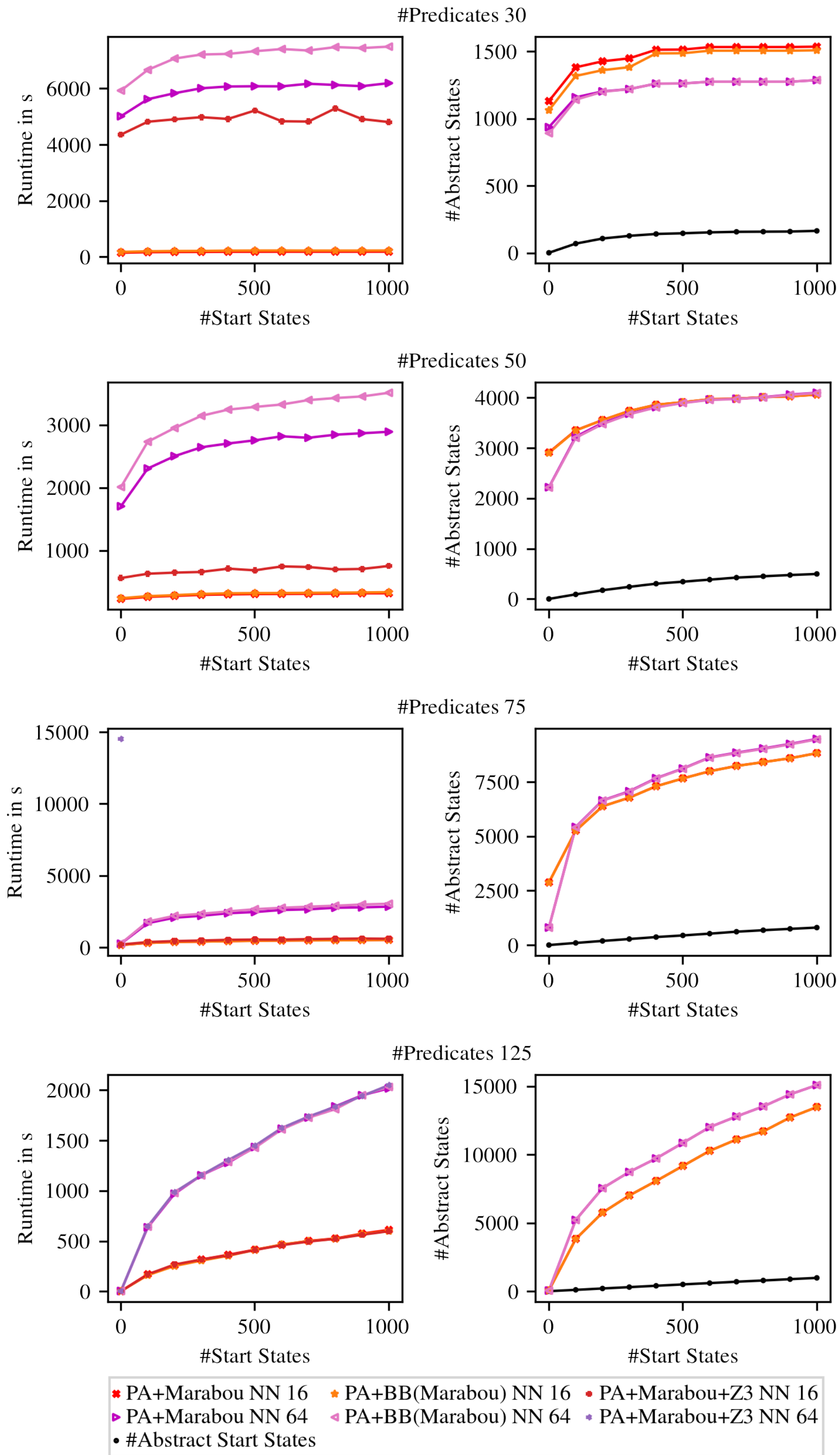

Figure 4: Time to compute the abstract state space (left) and the number of reachable abstract states (right) dependent on the number of concrete start states, for different methods (PA+Marabou, PA+BB(Marabou), PA+Marabou+Z3), NN policies with 2 hidden layers of size 16 and 64, and for predicate sets of different size $(30, 50, 75, 125)$. In the plots on the right, we also include the number of abstract start states.

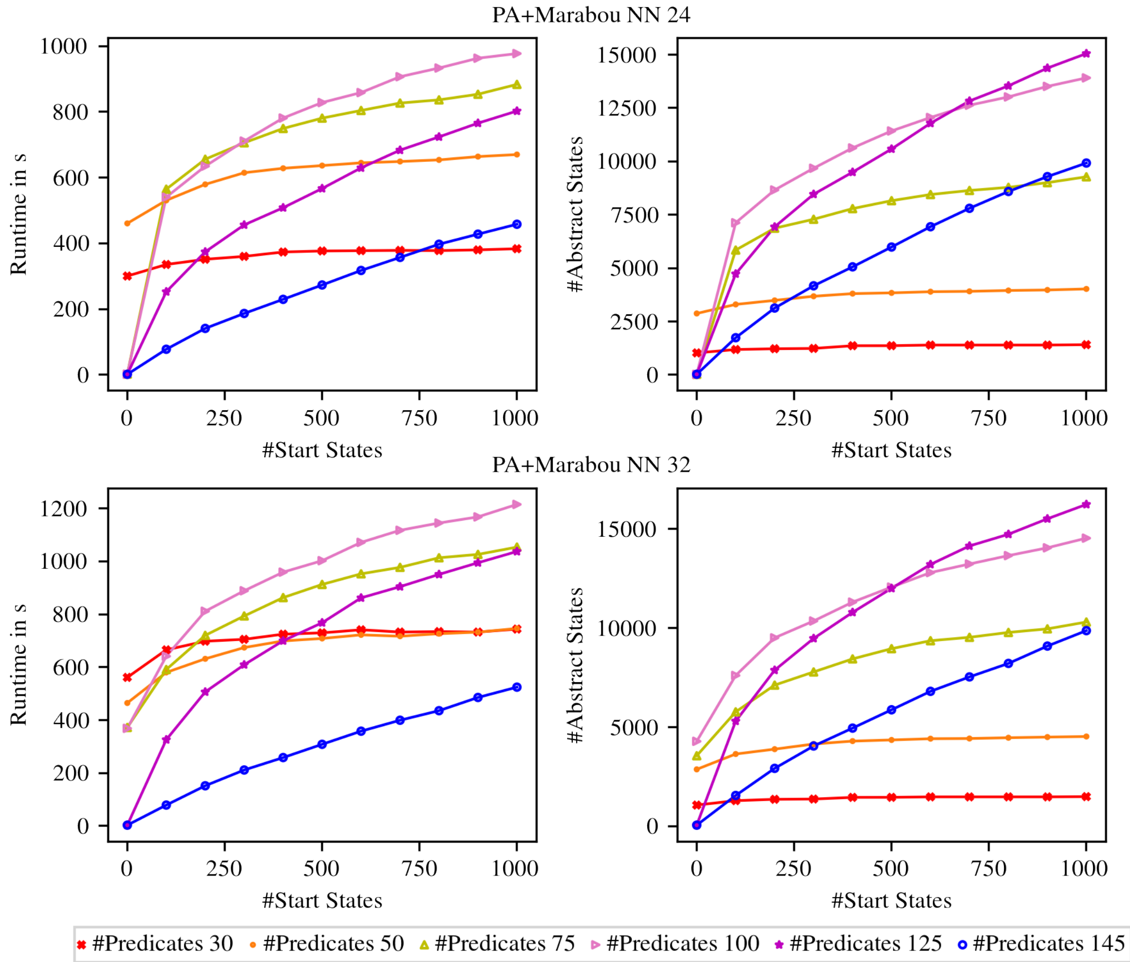

Figure 5: Time to compute the abstract state space (left) and the number of reachable abstract states (right) dependent on the number of concrete start states, for PA+Marabou, NN policies with 2 hidden layers of size 24 and 32, and for predicate sets of different size $(30, 50, 75, 100, 125, 145)$.